IN-61-
14727
P.29

Content Addressable Memory Project
NASA NAG-2 668

Semiannual Progress Report, March-August 1991

Rutgers University
New Brunswick, NJ 08903

J. Hall (PI) S. Levy (PI) D. Smith S. Wei K. Miyake
M. Murdocca
Laboratory for Computer Science Research
Department of Computer Science

October 1, 1991

CASI

1 Overview

This report describes the last six months progress on the Rutgers CAM Project.

The overall design of the system is complete at the architectural level and described in section 2. The machine, shown in Figure 1 on page 2, is composed of two kinds of cells; the CAM cells, which include both memory and processor, and support local processing within each cell; and the tree cells, which have a smaller instruction set, and provide global processing over the CAM cells.

We have completed a parameterized design of the basic CAM cell. The parameters are technology dependent and are concerned, not with the basic form of the cell, but such characteristics as the width of the data word within the cell. The instruction set for the CAM cells has been designed and is described in section 3. An instruction level simulator has been designed, implemented, and used to simulate algorithms on this architecture.

Progress has been made on the the final specification of the CPS and is described in section 5. We have a partial instruction level simulator for this component but we have not as yet settled on the final instruction set.

The gate level simulator described in section 4 is almost completed. It will be used to evaluate the design the details of both tree and CAM cells as well as the CPS.

The machine architecture has been driven by the design of algorithms whose requirements are reflected in the resulting instruction set(s). A few of these algorithms are described in section 6.

We have begun the design of a high level language, which not only will take advantage of the potential parallelism, but whose compiler will be an *expert (rule-based) system* that will take advantage of the associative properties of the CAM to support the compiling process. A discussion of our approach to the compilation process is contained in section 7.

2 Hardware Design

Figure 1 shows the Rutger's CAM architecture as a collection tree sitting over a set of CAM cell memories. This tree is composed of two types of nodes, leaf nodes termed CAM cells and internal nodes termed TREE cells. The two types of cells serve complementary functions within the architecture. CAM cells support local processing within each cell and its attached memory while TREE cells provide global processing for data collection, data movement, and parallel prefix operations over these same memories.

2.1 Supported Operations - a Functional Description

The supported operations fall into two major classes, local and global. Local operations function in a pure SIMD fashion: the same instruction is executed in each CAM cell using

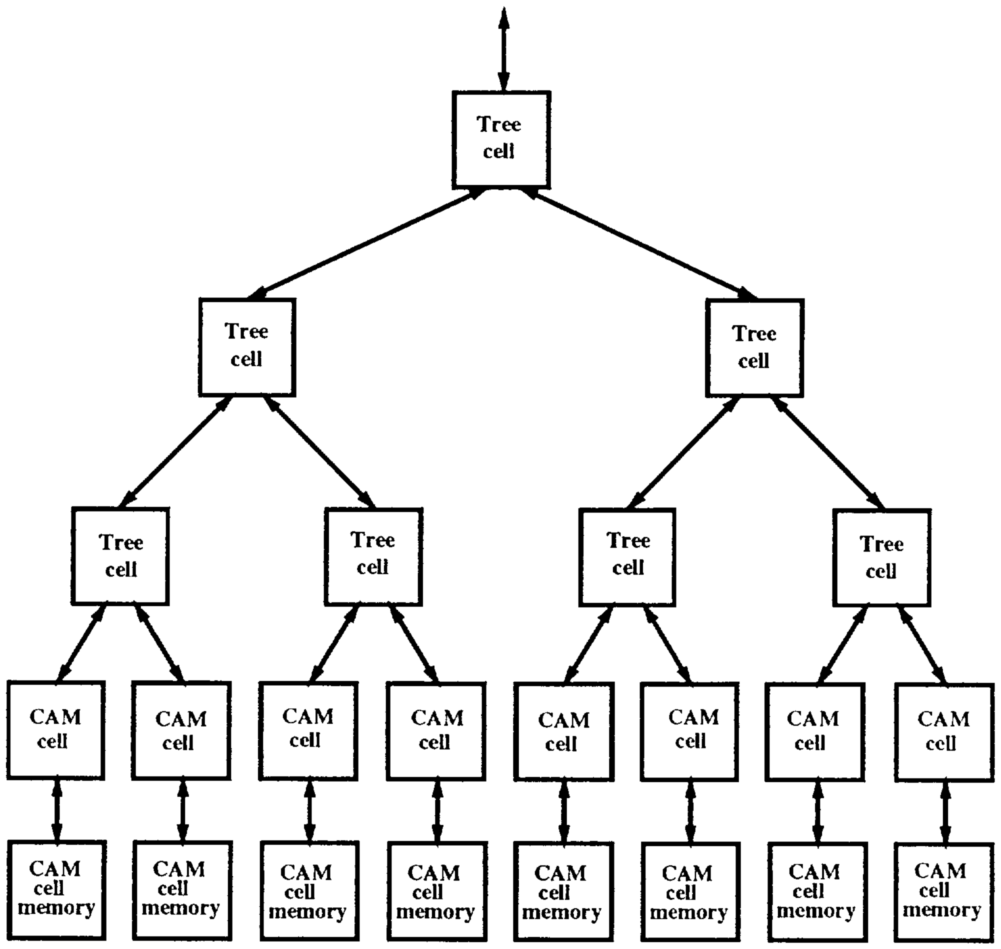


Figure 1: Collection tree composed of Tree and CAM cells

the same address in each CAM cell memory. They operate independently within each CAM cell and do *not* communicate between CAM cells. The global operations also operate in a SIMD fashion. In addition they execute the same instruction in each TREE cell[1] and through the use of these cells provide communication among the CAM cells and operations that are dependent on data from all CAM cells.

2.1.1 Local Operations

Local operations are performed independently within each CAM cell and its associated memory in a SISD manner. The only extension to the standard SISD model is the typical SIMD extension where each cell is enabled or disabled based on the setting of an activity bit.

Two examples of the use of activity control are shown in Figure 2. Both examples add 2 to each CAM cell: one shows the results when all cells are enabled while the other shows the results when some cells are disabled. A value of 1 in the activity bit indicates that the cell is enabled and will execute the instruction. A value of 0 in the activity bit indicates that the cell is disabled and will *not* execute the instruction. Notice that when a cell is disabled it maintains its previous state.

	Without activity control All cell enabled								With activity control Some cells disabled							
activity bit	1	1	1	1	1	1	1	1	1	0	1	1	0	1	1	0
initial state	2	3	4	5	6	7	8	9	2	3	4	5	6	7	8	9
final state	4	5	6	7	8	9	10	11	4	3	6	7	6	9	10	9

Figure 2: Local Add 2 on an 8 cell CAM

2.1.2 Global operations

Global operations can themselves be divided into two subclasses, unsegmented and segmented. Unsegmented operations consider CAM cells as a single contiguous segment and apply their functions to data in all CAM cells. Segmented operations partition the CAM cells into independent segments and perform the desired operation on each segment in parallel. As with local operations, global operations allow selective participation through activity control. Cells that are enabled send their data into the tree and receive their results from the tree. Cells that are disabled do not send their data into the tree and do not receive results from the tree but rather maintain their previous state.

Unsegmented operations: Unsegmented global operations accept their inputs from the CAM cells, carry out the necessary computation, and place the final result back in the

[1]The instruction executed by every CAM cell and the one executed by every TREE cell need not be the same.

CAM cells. Most of the global operations belong to the class of parallel prefix operations otherwise termed scan operations. These operations apply a binary associative operator $\diamondsuit$ with identity element ϕ to an ordered set $[x_1, x_2, \ldots, x_n]$ of n elements and return the ordered set $[\phi, x_1, (x_1 \diamondsuit x_2), \cdots, (x_1 \diamondsuit x_2 \cdots \diamondsuit x_{n-1})]$. Such scans are termed exclusive scans since element r_i of the result is dependent on x_j $\forall j < i$ but not dependent on x_i. The more common inclusive scan can be formed from the input x_i's and the exclusive scan r_i's by computing $x_i \diamondsuit r_i$ locally in each CAM cell.

In this architecture the CAM cells are connected using a binary tree and may therefore be considered ordered by any systematic tree traversal. The hardware supports preorder and reverse preorder orderings, also known as left-to-right and right-to-left; however, in this report only left-to-right orderings are described. The computations required by a scan are performed in parallel in the tree nodes with partial results communicated between neighbors. As with the local operations, unsegmented operations are extended to allow CAM cell data to selectively participate in a computation based on the setting of an activity bit in the CAM cell.

Two examples of an Unsegmented scan are presented in Figure 3. Both compute all partial sums, one with all cells are enabled and another with some cells are disabled. A value of 1 in the activity bit indicates that the cell is enabled and will participate in the computation. A value of 0 in the activity bit indicates that the cell is disabled and will *not* participate in the computation. Notice that when a cell is disabled it maintains its previous state.

	All cells enabled								Some cells disabled							
activity bit	1	1	1	1	1	1	1	1	1	0	1	1	0	1	1	0
initial state	2	3	4	5	6	7	8	9	2	3	4	5	6	7	8	9
final state	0	2	5	9	14	20	27	35	0	3	2	6	6	11	18	9

Figure 3: Unsegmented Partial Sum on an 8 cell CAM

Segmented operations: Segmented global operations operate much like their unsegmented counterparts; however, these operations partition the CAM cells in multiple independent segments. The segmentation is specified in segment bits that provide control over scan operations. A left-to-right scan uses these bits to determine if a CAM cell is the first element of a new segment and consequently whether a cell accepts data from its left neighbor. If the segment bit is 1 the CAM cell is treated as the first cell of a new segment and does not receive data from its left. If the bit is 0 the cell is a continuation of the current segment and receives data from its left neighbor.

Figure 4 present two examples of a segmented scan that computes all partial sums. As with unsegmented scans, one is an example without activity control while the other is an example with activity control.

	All cells enabled								Some cells disabled							
segment bit	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0
activity bit	1	1	1	1	1	1	1	1	1	0	1	1	0	1	1	0
initial state	2	3	4	5	6	7	8	9	2	3	4	5	6	7	8	9
final state	0	2	5	9	14	0	7	15	0	3	2	6	6	0	7	9

Figure 4: Segmented Partial Sum on an 8 cell CAM

2.2 Supported Operations - Hardware Implementation

Local and global operation modes are controlled by two bits stored in each CAM cell, the activity and segment bits. Local operations are dependent only on the activity bit while global operations depend on both bits. Instructions are executed under the control of these bits as determined by dedicated hardware in the CAM and TREE cells.

This section describes the hardware dedicated to computing the effect of segment and activity bits on an operation. The section is divided into three parts; local operations, unsegmented global operations, and segmented global operations.

2.2.1 Local Operations

Local operations make no use of the collection tree and as such reference only the activity bits. Each cell works independently of all other cells and updates its state based on the setting of its activity bit. Conceptually, the operation used for this purpose is an *enabled write* operation that writes the CAM cell's accumulator to the cell's memory if the activity bit is 1 and does not write if the bit is 0.

Since the SIMD nature of the architecture requires all cells to write to memory if any one writes to memory this conceptually simple operation is slightly more complex. The *enabled write* is implemented using a technique similar to a standard DRAM read/refresh cycle. The first step is to read from memory into the refresh register of each cell; the second is to overwrite the refresh register with the cell's accumulator in every with the activity bit set, and the third is to write the refresh register to memory in all cells.

Using this approach, memory accesses are independent of the state of a cell's activity bit and consequently performed by every cell. The selective overwrite of the refresh register causes the memory of an enabled cell to be updated and the memory of a disabled cell to be refreshed (i.e. state maintained). Implemented as described, the *enabled write* requires no more time than a typical DRAM read cycle.

2.2.2 Unsegmented Global Operations

These operations treat the CAM cells as one contiguous segment and use activity bits to determine if a cell will participate in the operation: a 1 indicates the cell will participate in the operation; a 0 indicates it will maintain state and not participate. Unlike its effect on local operations one CAM cell's activity bit can effect another CAM cell's memory. As noted in section 2.1.2 most global operations are parallel prefix (i.e. scan) operations. This section will describe how the hardware supports scan operations.

These operations require data to be processed in the collection tree. The technique we have adopted performs a scan in two phases; an up phase during which data is processed, stored, and propagated up the collection tree; and a down phase during which the stored data and a value injected at the root are processed and propagated down the collection tree. Each TREE cell is connected to its left child, right child, parent and contains one internal register.

During each phase of a scan, tree cells perform two parallel operations determined by the type of scan (i.e. up or down) and the operation broadcast to the cells. During the up phase each tree cell stores the data from its left child into its internal register, applies the specified operation to the data from its children, and routes this result to its parent. The down phase works similarly; each tree cell routes the data from is parent to its left child, applies the specified operation to the data from its parent and internal register, and routes the result to is right child. A syntactic description of this computation and communication is provided in figure 5. Note that as in the above description it is assumed that values contained in a cell's internal register, v_i, during the down phase were stored during the corresponding up phase of the operation.

Figure 6 provides a detailed example showing the data transmission and storage of the two phases of a plus scan without activity control(i.e. all cells participate). In this example each TREE cell is shown as a rectangle containing a smaller rectangle that represents its internal register. The up phase passes data from each leaf node to its parent where the required computation (in this case addition) and communication is performed. Values transmitted during this phase are represented by integers placed at at the top of an edge while the values stored in internal registers are represented by the integers in the small rectangles. The up phase is complete when the computation reaches the top (in this case when 44 is output from the root) and the down phase is initiated.

A value of zero is introduced at the root at the beginning of the down phase. In this phase data is passed down from parent to children. Values transmitted during this phase are represented by the integers at the bottom of each edge. This phase is complete when values have been transmitted to and stored by each leaf cell. Note that the value introduced at the root during a down scan (in this case 0) represents the result of applying the TREE cell operation to all data that precedes, in a left-to-right sense, the left most CAM cell.

When an unsegmented scan is performed with some cells disabled the computation and communication of data in the collection tree must change. One way to accommodate this change without altering the operation of the TREE cells is to locally preprocess the data in the CAM cells, then perform an unsegmented scan with all activity set to 1, and finally to locally postprocess the data in the CAM cells. This approach permits CAM cells to have the appearance of being disabled in an unsegmented scan without processing activity information in the collection tree; however, this approach does not generalize to allow segmented scans or

$$\text{up:}\quad \begin{array}{l} v_p \leftarrow v_l \Diamond v_r \\ v_i \leftarrow v_l \end{array} \qquad \text{down:}\quad \begin{array}{l} v_r \leftarrow v_p \Diamond v_i \\ v_l \leftarrow v_p \end{array}$$

Figure 5: Syntax of up and down phases of a scan

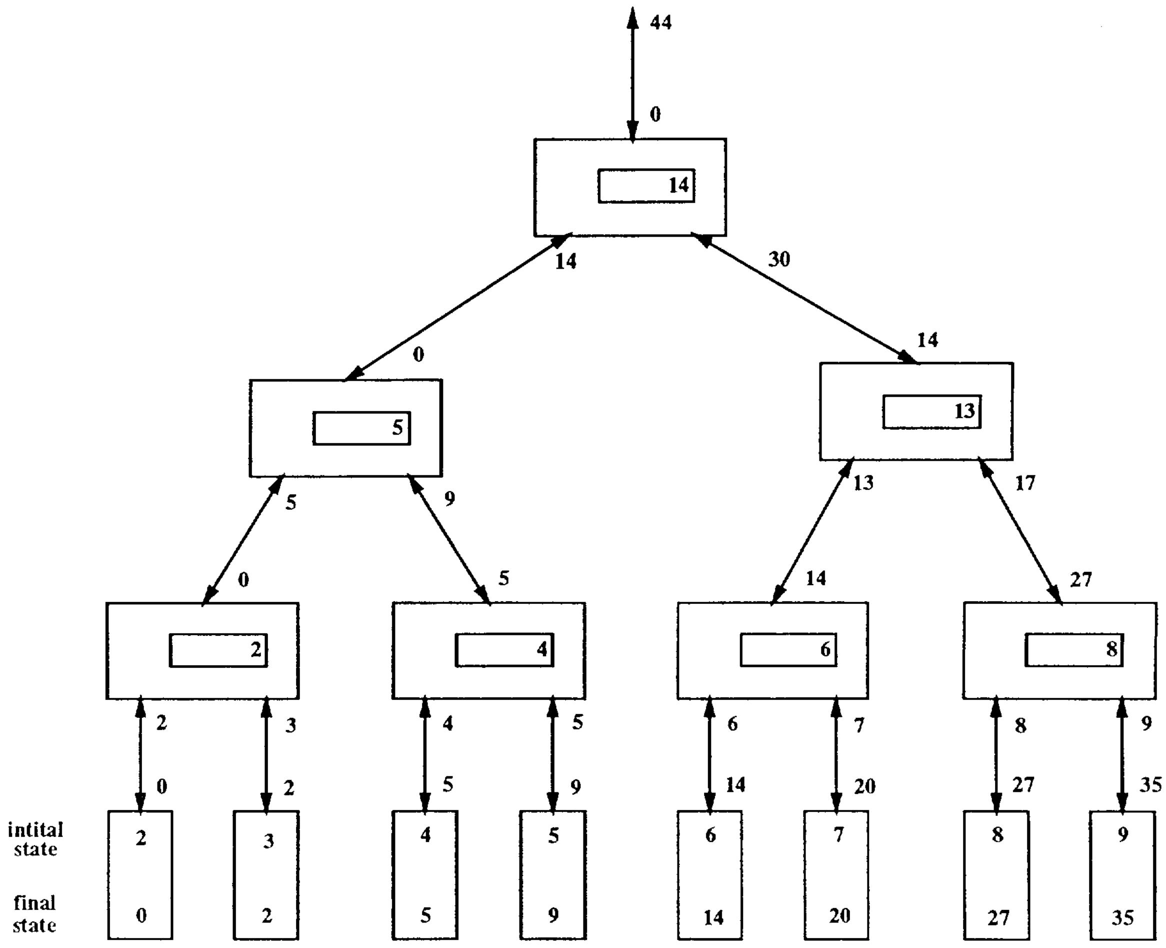


Figure 6: Computation and Communication for both phases of a plus scan

other global operations. An alternative approach, the one adopted for this architecture, is to pass activity control information up and down the collection tree. This approach requires a little additional hardware in each TREE node but provides uniform support for segmented, as well as unsegmented operations, with or without activity control. Section 2.2.3 gives a detailed explanation of how this is implemented.

2.2.3 Segmented Global Operations

Segmented operations require each TREE cell to be aware of the relationship between its two children. For example, during the up phase of a plus scan the data arriving from the children must be added together if the children belong to the same segment but must not be added if the children belong to different segments. The necessity for a TREE cell to react to segmentation information requires that the segment bits stored in the CAM cells be transmitted to and within the collection tree.

This architecture implements segment computation and communication with dedicated hardware in each TREE cell. The hardware accepts segment data from its two children, computes its own segment data, and passes this result to its parent. This data is encoded as a single bit that indicates if a new segment begins in the tree rooted at a TREE cell: a 1 indicates a new segment begins in the tree and a 0 indicates that no new segment begins in the tree. This representation provides a common semantics for segment information across CAM and TREE cells. CAM cells store this information in their segment bit while TREE cell compute this information as the or of their children's segment data.

Activity information must also be propagated within the collection tree; however, in contrast to segmentation information that only is passed up the tree, activity information must also be passed down the tree. The activity information is encoded as a 1 bit field that is conceptually attached to each data value passed in the tree. A value of 1 indicates that the data is significant and must be used in the computation while a value of 0 indicates that the data is irrelevant and should not be used.

For unsegmented scans the activity information could be handled in the same manner as segment information with the exception that activity information must also be propagated down the tree. However, this is not possible for segmented scans where activity information is dependent on segment information. Activity information, as was the case with segment information, can be given unified semantics that encodes activity in a single bit field that indicates if the associated data is relevant. Figure 7 shows the relationship between the activity and segment bits for segmented scans with activity control. Subscripts l and r denote information about the left child, right child, and parent of a node while superscripts u and d denote the attachment of activity information to the up and down phases of a scan.

Figure 8 shows the data paths that communicate the segment and activity information to a TREE cell. These cells compute their own segment and activity as specified in Figure 7 and transmit this information as indicated by the edges. The segment and activity control over the tree as a whole is determined by this hardware and the setting of the segment and activity bits in the CAM cells as well as an single activity bit, a^d, introduced at the root of the collection tree.

$$s \leftarrow s_l \vee s_r$$
$$a^u \leftarrow (\overline{s_r} \wedge a_l^u) \vee a_r^u$$

$$a_l^d \leftarrow a^d$$

$$a_r^d \leftarrow (\overline{s_l} \wedge a^d) \vee a_l^u$$

Figure 7: General left-to-right segment and activity

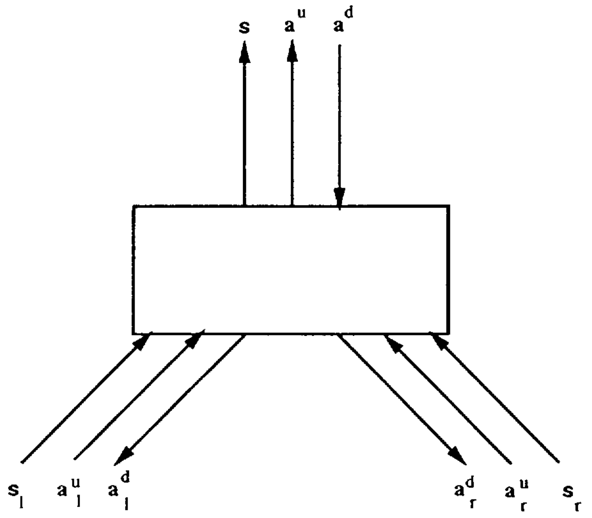


Figure 8: Communication paths for a Tree Node

Data Computation and Communication: Once each TREE cell has established its segment and activity information the data for the corresponding operation is processed and routed. Figures 9 and 10 show the computation performed by a TREE cell as a function of the activity and segment information during the up and down phases. Notice that a λ indicates that the value transmitted or stored will not be used in the computation. Since the knowledge that data is irrelevant to future computations is also transmitted in the activity and status bits the hardware treats these entries as *don't cares*.

Figures 7, 9, and 10 provide a complete description of how the TREE cells implement a general left-to-right segment scan with activity control. Scans without activity control as well as unsegmented scans are specialized version of the general scan and can be performed by setting the CAM cell's segment and activity bit as required.

s_r	a_l^u	a_r^u	v_i	v_p
0	0	0	λ	λ
1	0	0	λ	λ
0	0	1	λ	v_r
1	0	1	λ	v_r
0	1	0	v_l	v_l
1	1	0	v_l	λ
0	1	1	v_l	$v_l \Diamond v_r$
1	1	1	v_l	v_r

Figure 9: TREE cell Computation for up phase of unsegmented general scan

s_l	a_l^u	a^d	v_l	v_r
0	0	0	λ	λ
0	0	1	v_p	v_p
1	0	0	λ	λ
1	0	1	v_p	λ
0	1	0	λ	v_i
0	1	1	v_p	$v_p \Diamond v_i$
1	1	0	λ	v_i
1	1	1	v_p	v_i

Figure 10: TREE cell Computation for down phase of unsegmented general scan

3 Instruction Set

At present, the instruction set includes 192 instructions which can be divided into two categories:

1. *Vector (Tree) instructions (120):* The operation performed by each instruction is on the operands from all CAM cells. These instructions can be subdivided into the following groups:

   i) *Scan (48):* Each instruction performs one of "scan" (parallel prefix) operations such as arithmetic sum, logic AND/OR/XOR, and MAX/MIN. The scan operation can be done either left to right (start at the CAM cell of the lowest address) or right to left (start at the CAM cell of the highest address). Each operation can be also done under the activity control and/or segmentation control.

   ii) *Shift (24):* Each instruction performs one of "shift" operations such as shift, skip-shift (only active cells participate in the shift operation), rotate, and skip-rotate (only active cells participate in the rotate operation). The number of positions to be shifted is always one. The direction of a shift can be right to left (from higher address to lower address) or left to right (from lower address to higher address). The operations can be controlled by activity (mandatory for skip-shift and skip-rotate) and/or segmentation flags.

   iii) *Reduce (36):* Each instruction performs one of "reduce" (collection) operations such as arithmetic sum, logic AND/OR/XOR, and MAX/MIN. Like the scan operations, reduce operations can be done either left to right (placing the result on the right of a segment) or right to left (placing the result on the left of a segment). They can be also under activity and/or segmentation control.

   iv) *Broadcast (6):* Each instruction performs a broadcast operation either for all CAM cells or within each segment. These operations can be under activity and/or segmentation control. For the broadcast within each segment, it can be done either left to right (broadcasting from the left of a segment to the entire segment) or right to left (broadcasting from the right of a segment to the entire segment).

   v) *Others (6):* This group contains the instructions of setting an activity control flag (for all CAM cells or each segment) or loading a value to some particular location of each segment. These operations are always under activity and/or segmentation control.

2. *Scalar (CAM cell) instructions (72):* The operation performed by each instruction is on the operands in each individual CAM cell. The scalar instructions include arithmetic $+/-/\times$, logic AND/OR/XOR/NOT, 2's complement, bitwise AND/OR, bit-shift (left to right or right to left, and one or more bits), move, and comparison ($<, \leq, =, \geq, >, \neq$). Each instruction has two versions, one with activity control and one without activity control. If an instruction is under activity control, it will be executed only when the activity flag of the CAM cell is set to one.

The addressing modes for the instruction set are quite simple. For vector instructions, besides the optional activity control, segmentation control, starting value, and default value, each instruction may have one or two source operands and a destination address. The source operands come from memory (direct addressing) or an accumulator which is a general-purpose register. Since currently we assume that each CAM cell has only one general-purpose register, only one operand can be from the accumulator. We also assume that there is only one data path to memory. Thus, if an instruction has two source operands, one should be from the accumulator and the other should be from memory. The destination address can be either memory (also direct addressing) or accumulator. However, if an instruction has two source operands, its destination address must be the accumulator. Scalar instructions may also have two source operands and a destination address, besides the optional activity control and the number of bit positions to be shifted (for shift instructions). Similar to vector instructions, scalar instructions also have the one data path and one accumulator restrictions. However, a source operand in a scalar instruction can be an immediate value (i.e., immediate addressing mode).

4 Gate Level Simulation

A gate-level simulator is being created in C to help design and test the hardware implementation of the CAM and CPS chips. The simulator reads in VHDL-like hierarchical descriptions of modules, readies them for simulation, and simulates their operation.

A partial list of simulator commands is shown in figure 11.

```
source   <file>;              # read module descriptions from file
generate [module];            # create a module instance for simulation
destroy  [module];            # destroy the named module instance
read     <file>;              # read commands from file
run      [module];            # begin simulation on the instance
reset    [module];            # reset signal values in the instance
[time:] <signal> <- <value>;  # assign a value to the specified signal
[time:] show <signal>;        # print information about the signal
```

Figure 11: Partial list of simulator commands.

In figure 11, angle brackets (<>) refer to necessary arguments, while square brackets ([]) refer to optional ones. The simulator keeps track of a default module which is used when a module name is omitted. Commands are normally processed immediately, but the optional time argument may be used to assign and display signals while a simulation is running.

4.1 Module Definition

The simulator reads module definitions from named files, and builds gate-level descriptions of circuits. A module definition consists of the module name, followed by lists of the ports (or external signals), internal signals, and subcomponents of the module. Ports are the external connections to the module. Ports pass information between the module and other modules which may later reference it. Internal signals are signals created by the module. They are used to pass information between the module's subcomponents. Components are previously defined modules which perform subtasks.

Two types of modules are recognized by the simulator: *Primitive modules*, whose actions are defined by associated C-code, and *Composite modules*, which are composed from previously defined primitive and composite modules. Note that primitive modules have no internal signals or components.

The simulator also recognizes two types of signals: *Wires*, which take their input from a single primitive module, and *Busses*, which may take input from one or more special primitives called *tri-state drivers*. Tri-state drivers have a special output state which allows them to be connected to a bus but not actively drive the signal. During simulation, a bus may assume an unknown state if its drivers assert two distinct values on it.

An example of module definition syntax is shown in figure 12 on page 15, which describes an implementation of a three-input **AND** module. We note that there are better ways of constructing this module. In the example, the three-input **AND** gate is composed of two two-input **AND** gates, which are in turn composed of primitive nand and inverter modules (descriptions not shown). The description starts with the keyword *module*, followed by the module name. Following the module name are optional port, signal, and component lists, in order. A port list consists of the keyword *ports*, followed by lists of ports. Each list contains a list of signals, followed by one of the keywords *input*, *output*, or *inout*, which specifies the port type. The port types are used to help visualize the direction of data flow between modules. A (internal) signal list consists of the keyword *signals*, followed by a list of signal names. Internal signals are created by the module for communication between its submodules. A component list consists of the keyword *components*, followed by a list of component descriptions. A component description begins with a component name, followed by the module type of the component, and a list of signal names corresponding to the ports of the previously defined module, ending with a semicolon. The keyword *end* ends the current module description.

Some feasibility checks are performed on modules at the time of definition. Within a module each signal and component must be given a unique name[2], which is used for signal naming. Composite modules are also checked for compatibility with their components. Each signal in a component description must match the previously defined module's list of ports. This includes checking that busses are not connected to wires and that only one primitive module may output to a wire.

[2]The name of a signal is unique within a module; however, if a signal is available in different modules of a design it will have several equivalent names, one for each module in which the signal is available

The motivation behind the naming restrictions can be seen in figure 13 on page 15, which is a picture of the three-input **AND** of figure 12. The restrictions enable unique identification of every signal in the module. Signals which are not present in the top level module are referenced through their component, such as 'ab_and.z_bar'. This allows two instances of a module to use the same *local* name for two different signals but to distinguish one signal from the other by its position in the hierarchy. Signals 'ab_and.z_bar' and 'product.z_bar' in Figure 13 are examples of this naming convention. Signals known in multiple contexts will have multiple names each dependent on one of the contexts. An example of this is the signal 'ab', which is also known as 'ab_and.z' and 'product.x'.

4.2 Flattening

After definition, the selected module is *flattened* to ready it for execution. In the flattening process, each composite module recursively connects its components directly to the signals which affect them. By the end of flattening, primitive modules are connected directly to the signals which affect them, and composite modules are effectively hidden from the simulator. The shorter path between primitive modules and their signals results in faster simulations.

4.3 Simulation

The simulator is an event-driven simulator. In an event-driven simulation, module output values are only recomputed when one of its inputs have changed. This means that less extra computation is done compared to other simulation models, which results in shorter simulation times. In our simulation model, each primitive module takes a single unit of time for execution. Additional delays can be obtained by using special primitives called delay modules.

Because of the built-in delay for primitive modules, each signal can change only once during each time step. This allows us to process each time unit in three stages. First, all necessary changes are made to signal values. Second, the values of affected busses are evaluated. Finally, affected primitive modules are executed and their outputs are scheduled. This method permits the detection of race conditions in modules and busses (when two or more input signals change simultaneously, causing a short unstable state), and reduces the number of events needed to perform a simulation.

4.4 Future Enhancements

Future enhancements to the simulator include use of vector notation for groups of signals, notation for specification of iterative and recursive modules, improvement of user interface, and additional functionality for tracing and debugging simulations.

```
module and
  ports
    x y input
    z output
  signals
    z_bar
  components
    xy_nand nand x y z_bar;
    z_inv   inv z_bar z;
end

module three_input_and
  ports
    a b c input
    p output
  signals
    ab
  components
    ab_and  and a b ab;
    product and ab c p;
end
```

Figure 12: Composite definition of a three-input and module.

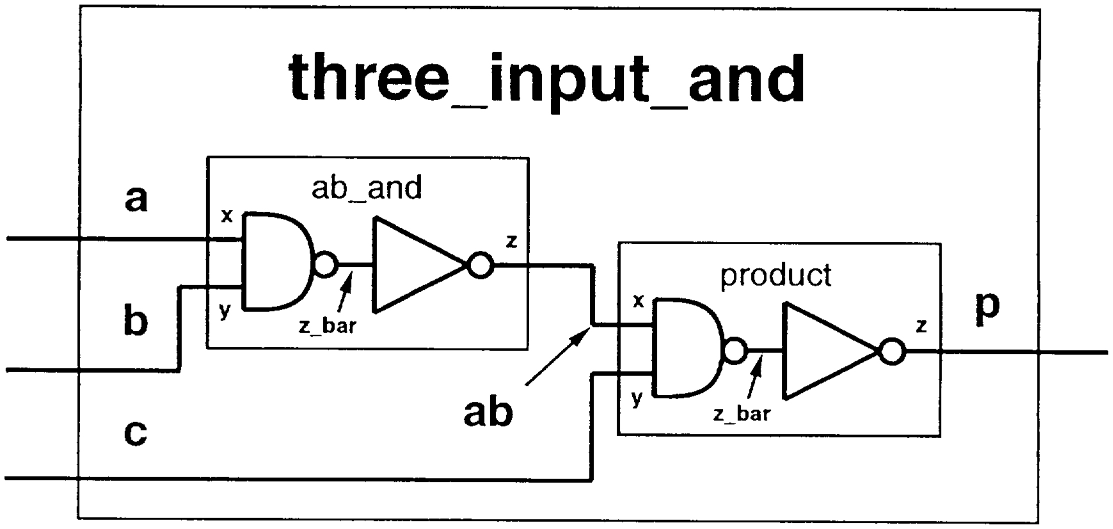


Figure 13: Graphical representation of the three-input and.

5 CPS Simulation

An additional C-based instruction level simulator has been written to test the developed architecture of the CPS portion of the CAM.The simulator reads a user supplied program written in a simple macro language and produces a trace of the executing program. The input program defines the architecture of the CPS processing elements (PEs), and provides the inputs and simulation parameters.

The section of code shown below is taken from a sample input file to the simulator:

```
DEFINE NUMBER_OF_PES 64
DEFINE NUMBER_OF_PE_REGISTERS 64

LEGENDCOMMENT Test message in position 1
LEGENDCOMMENT Test message in position 2

COMMENT This is a comment

LOAD CROSSBAR
11111111 10000000 00000000 00000000 00000000 00000000 00000000 00011000
00000000 01000000 00000000 00000000 00000000 00000000 00000000 00000000
[ 64 rows total ]
00000000 00000000 00000000 00000000 00000000 00000110 00000000 00000000

PRINT CROSSBAR

LOAD PE   0 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
          23 24 24 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44
          45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
LOAD PE   1 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
          23 24 24 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44
          45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
LOAD PE   63 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
          23 24 24 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44
          45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
PRINT PE_REGISTERS

STEP COMMAND_X
```

As shown at the top of the listing, the number of PEs and the number of PE registers are a few of the parameters that are defined by the user. The language uses a simple command+argument syntax which simplifies user modifications. The STEP command allows the user to create complex hardware descriptions by writing C code that directly emulates the hardware. There are well-encapsulated sections in the C source code for the simulator that simplify this process. The reason for taking this approach rather than creating a more general simulator that does not place a burden on the user to write C code is that it is difficult to capture the form of the CPS while the CPS is being developed. The STEP command takes an arbitrary number of arguments which are passed to the entry point in the simulator where the user has access to custom routines.

The PRINT command generates a PostScript graphical output file. A portion of the PostScript output is shown in Figure 14 on page 18. The crossbar settings are shown in graphical form, and are also provided in tabular form (not shown). A hexadecimal dump of the PE registers is created on each instance of the PRINT PE_REGISTERS command. The user can thus create an input file that contains data and commands to the simulator, and generate a graphical trace of the operation of the CPS that is convenient to produce on a PostScript output device.

The simulator supports step by step control of the CPS at the instruction level. Peek and poke capabilities are supported so that the user can observe execution of a CPS program and modify data as well as the crossbar switch settings.

The simulator and corresponding documentation are partially completed as of this report.

6 Function Level Simulation of Application Algorithms

The purpose of simulating algorithms is to find whether the defined instruction set is sufficiently powerful for implementation of typical parallel algorithms and to evaluate execution efficiency of these algorithms on our architecture using the instruction set.

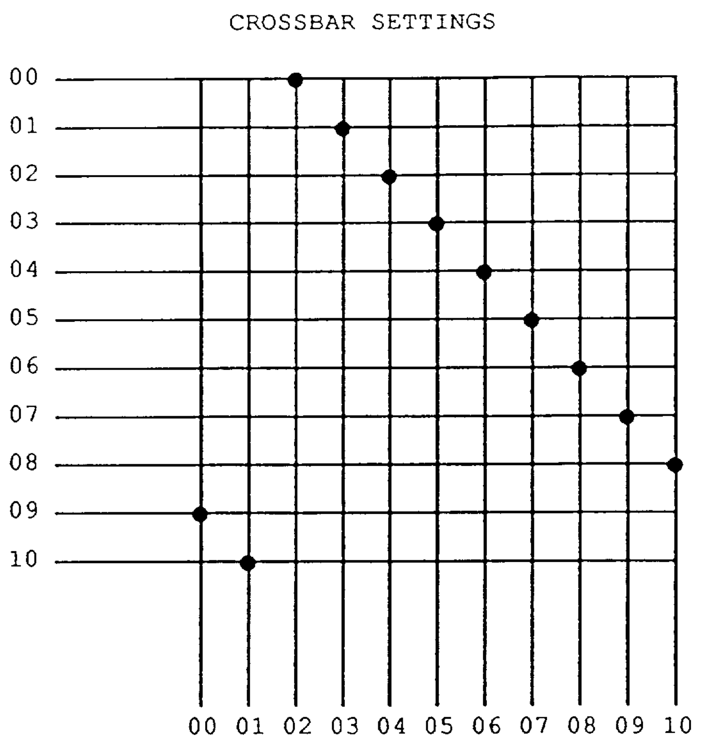
For the simulation, we assume that there are N (up to 64k) CAM cells, each containing a memory of 32 words which can be also used as the accumulator or control registers (e.g., activity control, segmentation, etc.). These numbers represent a much smaller memory than the architecture calls for, but we are running the simulation on a sequential machine at an estimated 100-million-to-one speed disadvantage compared to the actual CAM.

Simulation is done essentially at assembly language level. Each algorithm is written using these instructions and embedded into a C host program which provides I/O facility and control structures such as conditional or iterative statements (The functional capability of the C host program will be eventually provided by the central processing system (CPS)).

The simulator is implemented in C. Each instruction is implemented as a procedure or a function. Thus, each algorithm is realized by a series of procedure/function calls. In the following, we describe in detail the simulation of three algorithms.

6.1 Expression parsing

Problem 1: Given an expression and an operator precedence grammar, parse the expression. (i.e. create the corresponding parse tree.) The grammar is to be represented by left and right numerical precedence values for each operator. The ability to perform this basic



```
PE      Hex dump

00      00000000    00000001    3f3f0000    00000005
        00000008    00000009    7702fffe    0000000d
        00000010    00000011    00000341    00000015
        00000018    00000018    0000022a    0000001d
        00000020    00000021    0000003d    00000025
        00000028    00000029    00000004    00000035      . . .
        00000030    00000031    0000003d    0000003d
        00000038    00000039    0000000f    00000025

01      00000000    00000001    8f000ff0    00120000
        00000008    00000009    00004000    00000022
        00000010    00000011    ffa80000    3a320043
```

Figure 14: Postscript output of CPS Simulation

algorithmic task in parallel demonstrates the applicability of the CAM to the hoped-for area of symbolic manipulation.

The initial input data include the expression, type of each operator or operand, and left precedence and right precedence of each operator or operand. Each CAM cell holds an item (i.e., an operand or operator) as well as its type and left and right precedences. The type of each item is defined as follows:

```
Type = 1,  an operand;
     = 2,  an operator does not have any operand;
     = 3,  an operator has a left operand;
     = 4,  an operator has a right operand.
```

The left and right precedences of an operand are always 0. The operators with the lowest precedence have 2 for left precedence and 1 for right precedence, respectively. The operators with the second lowest precedence have 4 for left precedence and 3 for right precedence, respectively, and so on. The resulting parse tree is represented by a vector (each CAM cell holds an element). Each element of the vector indicates the address of its parent (0 means the root of the tree). Table 1 on page 19 gives an example of Problem 2. We can see that the root of the parse tree is operator '+' in CAM cell 2, operand 'A' in cell 1 and operator '+' in cell 6 are two children of the root, etc.

Address of CAM cells:	1	2	3	4	5	6	7
Expression:	A	+	B	*	C	+	D
Type of items:	1	2	1	2	1	2	1
Left precedence:	0	2	0	4	0	2	0
Right precedence:	0	1	0	3	0	1	0
Parse tree (Parents):	2	0	4	6	4	2	6

```
    +
   / \
  A   +
     / \
    *   D
   / \
  B   C
```

Table 1: An example of Problem 2: Parsing $A + B * C + D$.

Algorithm 1. Expression parsing.

Input: The expression, type of each item (TY, and precedences of each item (LP and RP).

Output: The parse tree represented by a vector, PA, containing the parent address of each item.

Method:

1. Initialization: Clear PA_i of every CAM_i.

2. Check if only one CAM cell has $PA = 0$. If it is, (i.e., the cell contains the root of the parse tree), output the parse tree and then stop. Otherwise, go to the next step.

3. For every CAM cell, say CAM_i, in which its TY_i is not 0, do the following:

i) Compare its LP with the RP of CAM_{i-2} (For the indexes of CAM cells, we do not count the CAM cells in which their TY's are 0). Save the result of the comparison in TMP_i. If $LP_i > RP_{i-2}$, $TMP_i = 1$, otherwise, $TMP_i = 0$.

ii) If $TY_i = 2$ or 4 (i.e., an operator does not have any operand or has a right operand) and $TMP_i = 1$, copy the address of CAM_i, i, to PA_{i-1} of CAM_{i-1}.

iii) Compare its LP with the RP of CAM_{i-2} again. Save the result of the comparison in TMP_{i-2}. If $LP_i < RP_{i-2}$, $TMP_{i-2} = 1$, otherwise, $TMP_{i-2} = 0$.

iv) If $TY_i = 2$ or 3 (i.e., an operator does not have any operand or has a left operand) and $TMP_{i-2} = 1$, copy the address of CAM_i, i, to PA_{i+1} of CAM_{i+1}.

v) Change TY_i: (a) If a CAM cell is holding an operand at this iteration ($TY_i = 1$), it will not participate in the next iteration, and so set its TY_i to 0. (b) If a CAM cell is holding an operator which has a left child, set its TY_i to 3. (c) If a CAM cell is holding an operator which has a right child, set its TY_i to 4. (d) If a CAM cell is holding an operator which has both left and right children, set its TY_i to 1, i.e., the operator will be considered as an operand at the next iteration.

vi) Change LP_i and RP_i: If an operator becomes an operand (TY_i becomes 1) at the last step, set its LP and RP to 0.

4. Count the number of CAM cells which have $PA = 0$. Go to Step 2. □

The number of instructions needed for each iteration is 66 (11 vector instructions and 55 scalar instructions), which is independent of the length of the input expression (i.e., the number of operands and operators). The number of iterations required is equal to the number of levels of the resulting parse tree minus 1. Therefore, if we assume that execution of each vector instruction takes a constant time, the complexity of the algorithm is $O(h)$, where h is the height of the parse tree. Since h is a function of the length of the input expression, say L, we can see that the best case is $h = O(\log L)$ and the worst case is $h = O(L)$.

6.2 Value update in a specified region

Outside the area of symbolic manipulation, we wish to evaluate the CAM as a general purpose machine for other tasks. This problem and the following one explore the ability of the CAM to handle problems it was not designed for. Thus we explore techniques for region labeling, a subpart of the vision architectures evaluation suite. The architectural tradeoffs made in the CAM are different from those in a machine intended for vision applications, so this constitutes a test of "the edges of its envelope".

First, we simply try to identify and operate on each element of a specified region in a one-dimensional vector.

Problem 2: Assume that each CAM cell has an integer number in its memory, $M_{src,i}$, $0 \leq i \leq N-1$. Now given a new value, k, and a vector of size N, V_i, in which each element corresponds to a CAM cell (i.e., $V_i \leftrightarrow M_{src,i}$) and all elements are 0's except that one element, V_j, is 1, change the value of $M_{src,i}$ to k if (a) $i = j$ or (b) $M_{src,i} = M_{src,j}$ and for all l, where $i < l < j$ or $j < l < i$, $M_{src,l} = M_{src,j}$.

Table 2 on page 21 gives an example of Problem 1. The associated algorithm is as follows.

```
Before:  M[src][i] =  1  1  2  2  2  2  3  3  2  2  2  2  2  1  1  2
Given:   V[i] =       0  0  0  0  0  0  0  0  0  1  0  0  0  0  0  0
         k =  6
After:   M[src][i] =  1  1  2  2  2  2  3  3  6  6  6  6  6  1  1  2
```

Table 2: An example of Problem 1.

Algorithm 1. Value update in a specified region.

Input: $M_{src,i}$ and V_i, where $0 \leq i \leq N-1$, and the new value k.

Output: The updated $M_{src,i}$, $0 \leq i \leq N-1$.

Method: It includes the following steps:

1. Find the value of $M_{src,j}$ where $V_j = 1$. Then, each CAM cell compares its $M_{src,i}$ with the value to see if they are equal.
2. Find the right boundary of the region, i.e., find the largest l, where $l \geq j$, such that $M_{src,l} = M_{src,l-1} = \ldots = M_{src,j+1} = M_{src,j}$ but $M_{src,l+1} \neq M_{src,j}$.
3. Find the left boundary of the region, i.e., find the smallest l, where $l \leq j$, such that $M_{src,l} = M_{src,l+1} = \ldots = M_{src,j-1} = M_{src,j}$ but $M_{src,l-1} \neq M_{src,j}$.
4. Broadcast the new value k to every CAM cell within the region and update its $M_{src,i}$, using the obtained boundary information to control. □

The total number of instructions used is 19 (5 vector instructions and 14 scalar instructions, not including I/O operations provided by the host program), which is independent of the number of CAM cells, N. Thus, the complexity of this algorithm depends on the complexity of vector instructions. If we assume that execution of each vector instruction takes a constant time, the complexity of the algorithm is $O(1)$.

6.3 Region separation by renumbering

Now, using the basic techniques from Problem 2, the real test. A true vision machine has 2-dimensional connectivity; the CAM is only 1-dimensionally connected.

Problem 3: Given an $M \times N$ matrix in which each element is an integer, renumber elements of the matrix in such a way that each region (a region consists of the adjacent elements with the same value) is assigned a unique value.

An application related to this problem is to use different colors to distinguish separate regions in an image. Table 3 on page 23 gives an example ($M = N = 16$) of Problem 3.

We try to solve (or partially solve) the problem by two steps, one for rows and the other for columns. At the first step, different regions in each row (i.e., the groups of the adjacent elements with the same value) are renumbered by an increasing sequence of nonnegative integers.

Then, in the second step, several columns (one column at a time) are selected and the regions of these columns are renumbered so that elements at different rows but in the same region have the same number. How many columns we have to select depends not only on the distribution of regions in a matrix but also on which columns are selected. If a column goes through every region in the matrix, selecting this column will be enough, i.e., it is the best case. The worse case can be that we have to select K columns, where K is proportional to the number of columns in the matrix, M. Also, sometimes there may not be sufficient information for selecting the appropriate columns, even though they exist. Therefore, column selection is heuristic. We believe that, given a characterization of the kind of pictures to be processed, heuristics can be selected to make the entire process proportional, on average, to the maximum number of regions touching any single horizontal line.

In the following, we give an algorithm which does row renumbering and one column renumbering. Note that if a region is not convex and a row/column crosses the region more than once, elements in the same region will be renumbered with different numbers (see the example in Table 4). Table 4 on page 24 gives the result of applying the algorithm to the input matrix given in Table 3, by selecting a particular column.

Algorithm 3. Region separation by renumbering.

Input: The $M \times N$ matrix to be processed, stored in memory location S of each CAM cell, and the selected column k.

Output: The processed matrix stored in memory location R of each CAM cell.

Method: The required $M \times N$ CAM cells are divided into M rows of N cells each, and their addresses are assumed to be two dimensional, i.e., $CAM_{i,j}$, where $0 \leq i \leq M-1$ and $0 \leq j \leq N-1$.

1. For each row of the matrix (i.e., $S_{i,0}$ to $S_{i,N-1}$, $0 \leq i \leq M-1$), do:
   i) Compare $S_{i,j}$ with $S_{i,j+1}$, $0 \leq j \leq N-2$, to see if they are equal. $S_{i,0}$ is also compared with the starting value provided by the instruction.
   ii) If $S_{i,j} \neq S_{i,j+1}$ (including the starting value and $S_{i,0}$), set $R_{i,j+1} = 1$, and 0 otherwise.
   iii) Do a prefix sum operation on each row of $R_{i,j}$. So, different regions at each row of $S_{i,j}$ have been renumbered in an increasing sequence of nonnegative integers

CAM cell:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Before	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
separation:	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0
	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0
	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0
	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0
	0	0	0	0	0	0	1	1	1	1	1	1	1	0	0	0
	0	0	0	0	1	1	1	1	1	1	1	1	1	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
After	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
separation:	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0
	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0
	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0
	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	2	2	2	2	2	0	0	0	0
	0	0	0	0	0	0	2	2	2	2	2	2	2	0	0	0
	0	0	0	0	2	2	2	2	2	2	2	2	2	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 3: An example of Problem 3.

Selected column: k = 8

CAM cell:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
After	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
separation:	0	0	1	1	1	1	1	1	1	1	2	2	2	2	2	2
	0	0	0	1	1	1	1	1	1	2	2	2	2	2	2	2
	0	0	1	1	1	1	1	1	2	2	2	2	2	2	2	2
	0	0	0	0	3	3	3	3	3	2	2	2	2	2	2	2
	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
	0	0	0	0	0	0	0	5	5	5	5	5	2	2	2	2
	0	0	0	0	0	0	5	5	5	5	5	5	5	2	2	2
	0	0	0	0	5	5	5	5	5	5	5	5	5	2	2	2
	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6

Table 4: The result of applying Algorithm 3 ($k = 8$) to the input matrix given in the last table.

stored in $R_{i,j}$.

2. For column k of the matrix (i.e., $S_{i,k}$, $0 \leq i \leq M-1$), do:

    i) Compare $S_{i,k}$ with $S_{i+1,k}$, $0 \leq i \leq M-2$, to see if they are equal. $S_{0,k}$ is also compared with the starting value provided by the instruction.

    ii) If $S_{i,k} \neq S_{i+1,k}$ (including the starting value and $S_{0,k}$), set $V_{i,k} = 1$, and 0 otherwise.

    iii) Do a prefix sum operation on the column of $V_{i,k}$. So, different regions at column k have been renumbered by an increasing sequence of nonnegative integers stored in $V_{i,k}$.

3. For each row of the matrix, if a region at the row contains the kth element, substitute all the $R_{i,j}$ in this region by $V_{i,k}$. □

The number of instructions needed for this algorithm is 26 (9 vector instructions and 17 scalar instructions), which is independent of the number of CAM cells (i.e., the size of a matrix). Thus, if we assume that execution of each vector instruction takes a constant time, the complexity of the algorithm is $O(1)$. Since the total time to process a matrix depends on the number of columns selected, the complexity of processing the matrix can be from $O(1)$ for the best case to $O(N)$ for the worst case.

Work on this algorithm is still in progress, particularly with respect to column selection and the coalescence of regions found from separate columns. However, it seems to give remarkably good efficiency on this problem for which the architecture would at first glance seem rather ill-suited. This gives us confidence in the usability of CAM in general-purpose applications.

6.4 Analysis

The simulation indicates that the defined instruction set is sufficient for the implementation of the three algorithms and so it can be also used to realize many other parallel algorithms. We have found that for the three algorithms, a small percentage of instructions are vector instructions (about 23% on the average). We tried to loosen the restrictions to one data path and one general-purpose register, i.e., allow two data paths to memory and two registers, and found that it saves a number of scalar instructions (about 11% of the total number of instructions on the average). We have also compared the situations in which the number of control registers is assumed to be different. We have found that using more than one register for activity control, we can also save scalar instructions (about 12% on the average).

7 High Level Language

Relative complexity of programming can be a drawback to any parallel architecture, but in a heterogeneous one it can assume critical proportions.

A major component in this complexity is the number of different representations that can be used for the various high-level data abstractions the problem is originally represented in terms of, and the different efficiencies of the algorithms that implement the high-level operations on those representations.

Most programming today still takes place at a very low level, that is to say that the programmer makes the choice of data-structure and algorithm and indeed does so at a very explicit and detailed level. This has, for the heterogeneous architecture, the dual infelicity of making the programmer tangle with a serious optimization problem, and facing the possibility that the solution to the problem may change drastically with a small perturbation of the overall code, requiring a complete rewrite. (In practice, the complete rewrite is rarely done, and the application proceeds with a suboptimal implementation.)

The solution is to have the code written in terms of the high-level abstractions in the first place, and have the system solve the optimization problem of the representations. Then the programmer can make his small change, and the system can emit completely different low-level code, but the programmer does not even need to know this.

This is largely the intent of object-oriented programming, but there are two major differences. First, in the typical object-oriented system, the choice of data structure is a simple default mapping of a single implementation to each high-level abstraction: there is no optimization of data-structures at all. This is to a large extent a result of the second difference, namely that the programmers are generally required to write their own abstractions, and it is hardly reasonable to expect them to write all possible implementations of the abstraction they are trying to encapsulate!

Rather than providing a (simplistic) mechanism for the programmer to encapsulate abstractions, our method is to provide a set of very general abstractions and the operations on them. The implementation actually contains many different algorithms corresponding to each operation, and has a number of different representations for each of the abstract types. The specific implementation of a given program can then be formulated as an optimization over all the different implementations implied by the choices.

For example, consider a CAD program where there is an operation to change the color of some object on the screen, and an operation to manipulate a collection of subparts as a single part. Both these operations require the manipulation of a connected component of an abstract graph, of object abstractions on one hand and of pixels on the other. In both cases the required components can be found by the transitive closure of the appropriate adjacency relation, but in practice they would never be implemented the same way.

In implementations on conventional machines, graphs are commonly represented as pointer structures; this is the representation assumed by most of the standard texts for discussing the complexity of graph algorithms. Parallel processing texts tend to use a bit-matrix representation (or a numerical matrix for graphs with weighted edges). In practice many cases where what is being done could be described by graph operations, such as the screen fill mentioned above, the representation is highly customized, for example an array of node values where the edges are an implicit mesh connection.

Beginning with each of the possible representations for the graph, there are several different algorithms that could be used to perform the operations needed. Many of these present the option of producing the result in different ways, i.e. as a modification of the original datastructure, as a new datastructure, or as a representation of the abstract result in a different datastructure.

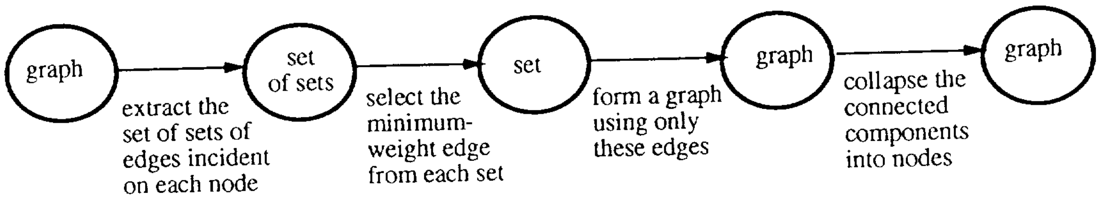
There are two major thrusts to our high-level language research. The first is to identify and implement a sufficient range of high-level abstractions, i.e. arrays, trees, graphs, sets, relations, etc, to cover a wide variety of useful programs. The implementations will not be simply a particular representation of each object, but as many different representations as resources and ingenuity permit. Each representation will be parameterized to allow an estimation of its cost in terms of time, space, communications bandwidth, etc.

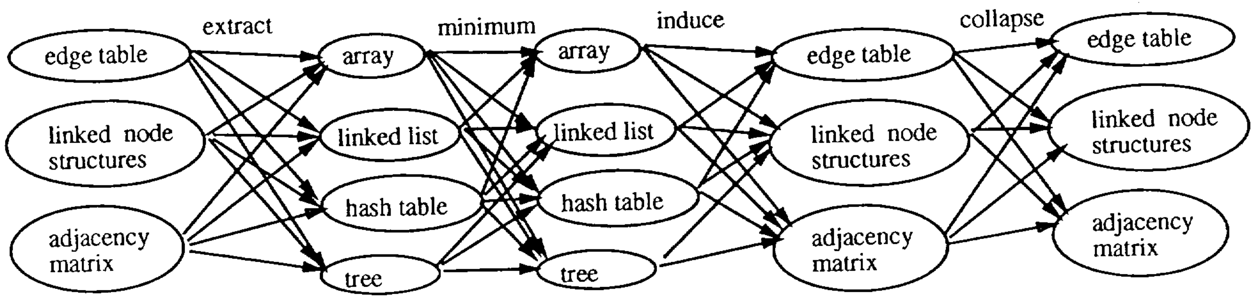
The second thrust is, given a program specified in terms of the high- level primitives, reduce it into an efficient implementation. This is more straightforward than the first part. We take the dataflow graph of the high-level program and "blow it up", expanding each object node into a cluster of representation nodes, and then replacing each operation edge in the original graph and replacing it with a whole subgraph of representation conversion–operation–representation conversion sequences corresponding to all the ways we have implemented which can convert some form of the input object to some form of the result object.

The resulting graph, although large, exceeds the original program by only a constant factor, which is related to the square of the number of representations that have been implemented. There are (surprise!) a number of representations and several efficient algorithms for finding the least cost path through such a graph. This is the basis for choosing the representations and algorithms in the compilation process.

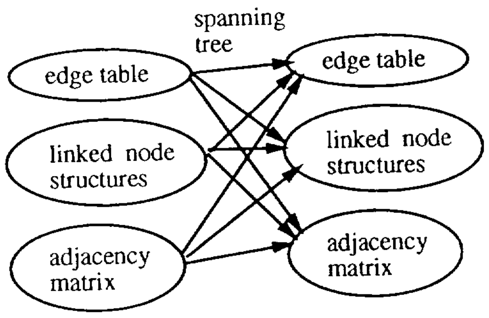
The compiler itself will obviously constitute a symbolic processing program of no small proportions, equivalent to a fair-sized expert system. Thus, development of the techniques necessary to implement the compiler efficiently on the CAM dovetail nicely with the task of developing the CAM as a platform for symbolic processing. For example, the production of a parameterized library of algorithms for the compiler to work from, can be seen as a formalization and rigorization of building a programming methodology for the CAM, which has been a priority through out the project.

Abstract operations for one iteration of a minimal spanning tree algorithm.


graph
extract the set of sets of edges incident on each node
set of sets
select the minimum-weight edge from each set
set
form a graph using only these edges
graph
collapse the connected components into nodes
graph



The simple graph for an abstract program gives rise to a "blown up" graph of possible implementations. An optimal path through the latter is then found.



The graph can be reduced by precomputing the optimal path from each input node to each output node. The reduced graph can then be used as a primitive.